Robust and reliable reconfiguration of cloud applications

Francisco Durán, Gwen Salaün

Robust and Reliable Reconfiguration of Cloud Applications

Francisco Durán[a], Gwen Salaün[b]

[a] *University of Málaga, Spain*
[b] *University of Grenoble Alpes, France*

Abstract

Cloud applications involve a set of interconnected software components running on remote virtual machines. The deployment and dynamic reconfiguration of cloud applications, involving the addition/removal of virtual machines and components hosted on these virtual machines, are error-prone tasks. They must preserve the application consistency and respect important architectural invariants related to software dependencies. In this paper, we introduce a protocol for automating these reconfiguration tasks. In order to ensure its correctness and robustness, we implement the protocol with the support of the Maude system for rapid prototyping purposes, and we verify it with its formal analysis tools.

Key words: Cloud Computing, Dynamic Reconfiguration, Rewriting Logic, Model Checking

1. Introduction

Cloud computing aims at delivering resources and software applications on demand over a network, leveraging hosting platforms based on virtualization, and promoting a new software licensing and billing model based on the *pay-per-use* concept. For service providers, this means the opportunity to develop, deploy, and possibly sell cloud applications everywhere on earth without investing in expensive IT infrastructure. Cloud computing is at the crossroads of several recent computing paradigms such as grid computing,

Email addresses: duran@lcc.uma.es (Francisco Durán), gwen.salaun@imag.fr (Gwen Salaün)

peer-to-peer architectures, autonomic computing, utility computing, etc. It allows users to benefit from all these technologies without requiring a deep expertise in each of them. In particular, autonomic computing is convenient for automating specific tasks such as the on-demand provisioning of resources or the elasticity of the application for facing peak-load capacity surge. Automation reduces user involvement, which speeds up the process and minimizes the possibility of human errors.

Many private and public clouds have emerged during the last years, offering to users a range of different services at SaaS, PaaS and IaaS[1] levels. In general, a cloud application may be seen as a distributed application composed of a set of virtual machines (VMs) running a set of interconnected software components. Such applications benefit from several services provided in the cloud, such as database storage, virtual machine cloning, or memory ballooning. To deploy their applications, cloud users need first to provision and instantiate some virtual machines and indicate the software components to be run on them. Once these applications are deployed, some reconfiguration operations may be required, such as instantiating new VMs, dynamically replicating some of them for load balancing purposes (elasticity), destroying or replacing VMs, etc. However, setting up, monitoring, and reconfiguring distributed applications in the cloud are complicated tasks because software involves many dependencies that oblige any change to be made in a certain order for preserving application consistency. Moreover, some of these tasks can be executed in parallel for execution time and performance optimization, but again this cannot easily be achieved manually. Thus, there is a need for robust protocols that fully automate reconfiguration tasks on running applications distributed across several VMs. The design of these reconfiguration mechanisms is complicated not only due to the high level of parallelism inherent to such applications, but also because they must preserve important architectural invariants at each step of the protocol application, *e.g.*, a started component cannot be connected to (and then possibly using) a stopped component. Failures can also occur and should be properly treated.

In this paper, we present a fully automated, decentralized, robust, and reliable protocol which aims at reconfiguring at runtime cloud applications

[1]The different types of Cloud Computing services are commonly referred to as Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS).

consisting of a set of interconnected components hosted on remote VMs. We consider several kinds of reconfiguration operations, namely addition and suppression of bindings, components, and VMs. When configuring an application, the protocol is able to instantiate VMs, effectively connect the components as required, and start these components respecting their functional dependencies. When removing parts of an application, the protocol needs to stop and disconnect components in a certain order for preserving the architecture consistency. For instance, since we never want a started component to be connected to a stopped component, in order to stop a component, we must previously ask their client components (components bound to that component) to unbind, and we can stop the component only when they have all done so. This supposes a backward-then-forward propagation of messages across VMs composing the application, along bindings connecting components on mandatory required services.

The protocol presented in this article supports *external* failures, which are failures of the execution environment (*i.e.*, the virtual and physical infrastructure) and the management system itself. This article focuses on failures on VMs—we will use failure or VM failure indifferently in the remainder of this article. Applicative failures, which are specific to a given application, or failures of the cloud manager are not supported. Moreover, we only consider permanent failures: if a transitory failure is detected, it will be treated as a permanent failure.

Due to the high degree of parallelism inherent to the applications to be reconfigured, the design of these reconfiguration mechanisms is very complicated and would have not been possible without the support of formal techniques and tools. Particularly, we have implemented the reconfiguration protocol using the rewriting-logic-based Maude language [10]. We have chosen Maude because it relies on a declarative style, which simplifies program writing and rapid prototyping. Maude is also very expressive for developing concurrent programs involving nondeterministic computations. Another advantage of the Maude system is that it is equipped with several formal analysis techniques and tools, which turn out to be very useful when designing highly parallel, and therefore error-prone, protocols. As for verification of the protocol, we used the Maude interpreter and its reachability analysis tool in early stages for correcting simple errors. Then, we verified a number of crucial properties on the protocol with Maude's LTL model checker, which helped us to detect and correct subtle bugs in boundary cases that would have been very difficult to identify otherwise.

The main contributions of this work are the following:

- We propose and design a robust protocol for runtime reconfiguration of cloud applications consisting of interconnected components hosted on several VMs.
- We implement the protocol using Maude, which results in a formal specification giving a precise rewriting logic semantics to our reconfiguration protocol.
- We validate our protocol through extensive application on a large number of component assemblies and reconfiguration scenarios using Maude verification tools.

A first version of this article was published in [11] and is extended here as follows:

- We have provided better and more extended descriptions of the problem and the solution, as well as a better contextualization of it;
- It is the first time that a description of the Maude prototypical implementation of our protocol, as well as its verification using Maude's analysis tools, is published;[2]
- The algorithm has been extended so that now it handles VM error situations, that is, if a VM fails, the algorithm is able to restore a state in which invariants are re-established;
- The discussion on related work has been updated and enhanced to give a larger and more up to date view and comparison with existing results.

The paper is structured as follows. We present the reconfiguration mechanisms in Section 2, and its Maude implementation and verification in Section 3. We review related work in Section 4 and we conclude in Section 5.

[2]The complete Maude implementation and some results on its accompanying analysis are available on the web at [1].

2. Reconfiguration Protocol

The reconfiguration protocol is presented in this section. Section 2.1 presents first the application model on which the algorithm applies, its main features in Section 2.2, and the protocol participants in Section 2.3. Sections 2.4 and 2.5 present the protocol mechanisms by describing, respectively, the activities carried out by the cloud and VM managers. Section 2.6 introduces the part of the protocol dedicated to failure recovery. We finally illustrate how the protocol works in practice through some sample scenarios of reconfiguration in Section 2.7.

2.1. Application Model

For the sake of comprehension, we abstract away from several implementation details such as IP addresses or configuration parameters. Thus, an application model consists of a set of VMs. From a functional point of view, these VMs do not play any role *per se*, but each of them hosts a set of components, where resides the functional part of the application. A component can be in one of these two states: *started* and *stopped*. A component can either provide or require services. This is symbolized using ports: an *import* represents a service required by a component and an *export* represents a service provided by a component. An import can be *optional* or *mandatory*. An import is *satisfied* when it is connected to a matching export and the component offering that export is started. Such a connection is called a *binding*. A component can import a service from a component hosted on the same VM (local binding) or hosted on another VM (remote binding). A component can be started, and then be fully operational, when all its mandatory imports are satisfied. A component can be fully operational even if its optional imports are not satisfied.

We will use as running example a typical three-tier Web application (Figure 1). Although this is a simple example, it shows several kinds of dependencies and allows us to illustrate our algorithm on interesting cases in a reasonable amount of space. VM1 hosts two components: a front-end Web server (Apache) and a profiling component. VM2 hosts an application server (Tomcat) and an object cache component. VM3 corresponds to the database management system (MySQL). These components are connected using local or remote bindings. These bindings can involve optional imports (o in the figure) or mandatory imports (m).

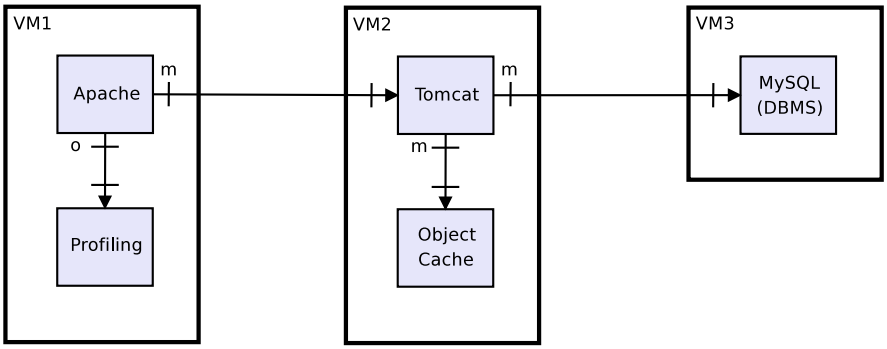

Figure 1: Example: A Web Application Model

2.2. Protocol Main Features

Our reconfiguration protocol exhibits four main important design features, namely, it is fully automated, decentralized, robust, and reliable.

Each VM is equipped with a VM manager in charge of *automating* the reconfiguration tasks at the VM level.[3] All VM managers work without any human intervention. A cloud manager (CM) posts reconfiguration operations that can be given by a cloud user or encoded into a scripting language. Thus, the CM does not necessarily require the presence of a human being for interacting with the running system and application.

VM managers are in charge of starting/stopping their own components and *no centralized manager* is used for that purpose. The protocol is also *loosely-coupled* because each VM manager does not have a global view of the current state of the application and particularly of the other VMs. Yet the VM managers need to exchange information in order to connect bindings on remote components or to let certain components know that other (partner) components have started or stopped. The only way to exchange necessary information for the component start-up/shutdown is to interact via asynchronous message passing. Each VM is equipped with two FIFO buffers, one for incoming messages and one for outgoing messages. VMs interact in a point-to-point fashion (no broadcast or multi-way communication). This solution is standard in distributed systems, and avoids the use of bottleneck

[3]We distinguish in the rest of this paper a VM, which is a software implementation of a physical machine, and a VM manager, which is the piece of software embedded on a VM in charge of applying the reconfiguration tasks on that VM.

centralized servers or communication media (*e.g.*, a publish-subscribe messaging system [2]), which limit the parallelism induced in the distributed system by transforming it somehow into a centralized one.

The protocol is *robust* in the sense that, during its application, some important architectural invariants are preserved, *e.g.*, all mandatory imports of a started component are satisfied (*i.e.*, bound to started components). These invariants are crucial because they ensure that component assemblies are well-formed. Therefore, they must be preserved during the whole lifetime of the application and at any step of the reconfiguration protocol execution.

This protocol is *reliable* because it is able to detect VM and network failures occurring during the reconfiguration process. When such a failure occurs, the protocol informs the remaining VMs of what has happened to make the system restore a consistent state. The protocol supports multiple failures and always succeeds in restoring consistency of the application in presence of a finite number of failures.

It is worth observing that when a VM fails, some invariants may temporarily be unsatisfied. Upon the occurrence of a failure, the algorithm proceeds deactivating the minimum number of components so that the invariants are eventually re-established (see Sections 2.6 and 3.2).

2.3. Architecture

The reconfiguration protocol involves a CM and a set of VM managers. The CM guides the application reconfiguration by instantiating/destroying VMs and requesting the addition/removal of components/bindings. Each VM in the distributed application is equipped with a VM manager that is in charge of (dis)connecting bindings and starting/stopping components upon VM instantiation/destruction operations posted by the CM. Communications between participants (CMs and VM managers) are achieved asynchronously via FIFO buffers. When a participant needs to post a message, it puts that message in its output buffer. When it wants to read a message, it takes the oldest one in its input buffer. Messages are transferred at any time from an output buffer to its addressee's input buffer. Buffers are unbounded, but the protocol does not involve looping tasks that would make the system infinitely send messages to buffers.

Figure 2 depicts a sample system with a CM and two VMs, and shows how they exchange messages through their buffers (dashed lines). More precisely, when the CM, for instance, needs to send an output message to VM1, it first adds it to its output buffer. The message is then transferred from CM's

output buffer to VM1's input buffer. The VM1 manager can finally consume this message from its input buffer.

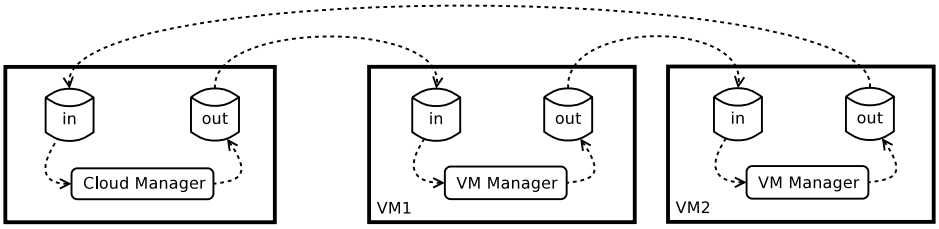

Figure 2: Architecture

2.4. Cloud Manager's Part

The CM submits reconfiguration operations to the running application and keeps track of the activation state of the deployed VMs and components. We consider the following reconfiguration operations:

- instantiation/destruction of a VM,
- addition/removal of a component on/from an existing VM, and
- addition/suppression of bindings.

In order to ensure a correct execution of the protocol, the CM validates the operations before applying them, *e.g.*, a VM is destroyed only if it was previously instantiated, a new binding is added only if both ports exist in the application, or a binding is added only if it does not form a cycle along mandatory imports. Our reconfiguration mechanisms are triggered by the execution of a sequence of such operations posted by the CM for, *e.g.*, maintenance or elasticity purposes [23].

The protocol works applying *up* and *down* phases. A phase has a coarse-grained granularity compared to atomic reconfiguration operations introduced above. An *up* phase corresponds to a set of reconfiguration operations dedicated to start-up operations (*e.g.*, VM instantiation or binding addition). When the CM instantiates a VM, it creates an image of this VM and the VM starts executing itself. When a CM adds a set of required bindings to the running application, it submits messages to all VMs impacted by these

changes, that is, all VMs hosting components involved in those bindings. These messages come with some configuration information necessary to the VM manager for binding purposes. In contrast, a *down* phase involves shutdown operations only (*e.g.*, VM destruction or binding removal). When the CM decides to destroy a VM, it sends a message to that VM. A VM destruction message implies the destruction of all bindings on components hosted on that VM. The CM also keeps track of the current activation state of all VMs running in the system (instantiated VMs and whether they are started or not). A VM is declared *started* when all components on that VM are started; or *stopped* otherwise. Figure 3 summarizes the CM lifecycle where we distinguish reconfiguration operations posted by the CM (solid lines) and messages received from the VM managers (dashed lines).

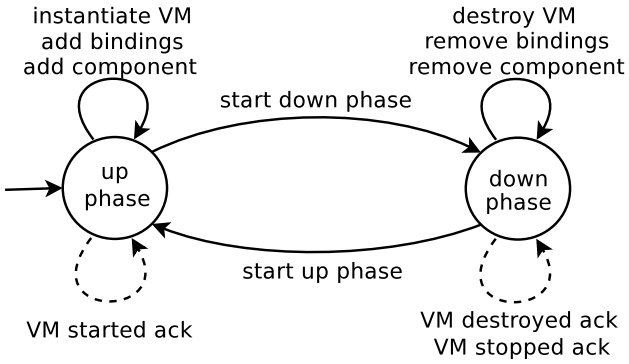

Figure 3: Cloud Manager Lifecyle

The up/down phases are applied alternatively in sequence, the CM being responsible for initiating a new phase. The application of these phases is therefore completely transparent for the user. It is worth noting that the alternative application of the phases by the CM implies some synchronization points. Before starting a new phase, the cloud manager waits for *ack* messages from the VMs involved in the former phase. When the cloud manager has received all these messages, it can initiate the new phase. Nonetheless, between two phases, the protocol is fully distributed and does not work using any centralized control. Likewise, keeping track of some information about VMs does not impact the distributed execution of the protocol, it is similar as logging information into a database.

We give in Figure 4 an example of scenario where first all VMs are instantiated and required bindings are added (Bds corresponds to the set of

bindings in Figure 1). Then, we decide to remove the MySQL component for replacing it by a new version. Finally, we add this new component (MySQL') on VM3 and add a binding (Bd') connecting the Tomcat component to the new MySQL component.

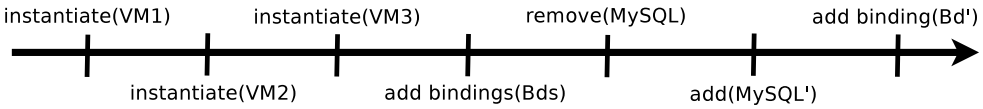

Figure 4: Web Application: Example of Reconfiguration Scenario

Figure 5 shows how the CM executes this scenario by applying successive up and down phases.

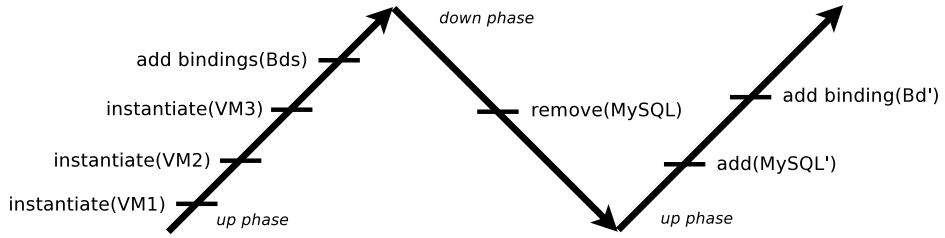

Figure 5: Web Application: Up/Down Phases for the Reconfiguration Scenario

2.5. VM Manager's Part

The VM manager of each VM starts its activity when the CM instantiates its VM. A VM manager is in charge of binding, unbinding, starting, and stopping components in its VM. In the rest of this section, we present the two most general reconfiguration operations, namely the instantiation and the destruction of a VM.

Binding and start-up. When a VM is instantiated, it is created with a number of components, each of which is initially off. Figure 6 shows how a newly instantiated VM proceeds in order to bind its ports and start its components. After its instantiation (❶), the VM manager can immediately start a component without imports or with optional imports only (❷). If a component involves mandatory imports, that component can only be started when all its mandatory imports are satisfied, *i.e.*, when all these imports are bound to started components. When a component is started, its VM

manager informs the VM managers of all remote components using it by sending *component started* messages (❸). If all components of a VM are started, its VM manager sends a message to inform the CM (❹), otherwise it starts reading messages from its input buffer (❺):

- If a VM receives from the CM some binding requests (for both local and remote bindings), the manager first establishes local bindings (❼). Handling of remote bindings is initiated on the export side: when an export of one of its components is involved in a binding, a VM manager sends a message (❽) with its export connection information (*e.g.*, IP address) to the VM hosting the other component (import side).
- If the VM receives a remote binding message, this means that an import of one of its components is involved in a binding. Upon reception of that message, the VM manager makes the binding effective (❻).
- Every time a *component started* message is received, the VM manager checks if the corresponding components can be started (❷). Each VM manager keeps the states of its partner components.

Note that the start-up process implies a propagation of *started* messages along bindings across several VMs. Local bindings are handled directly by VM managers, without additional messages with other VMs. The algorithm checks for cycles of bindings over mandatory ports, thus ensuring the termination of the start-up process.

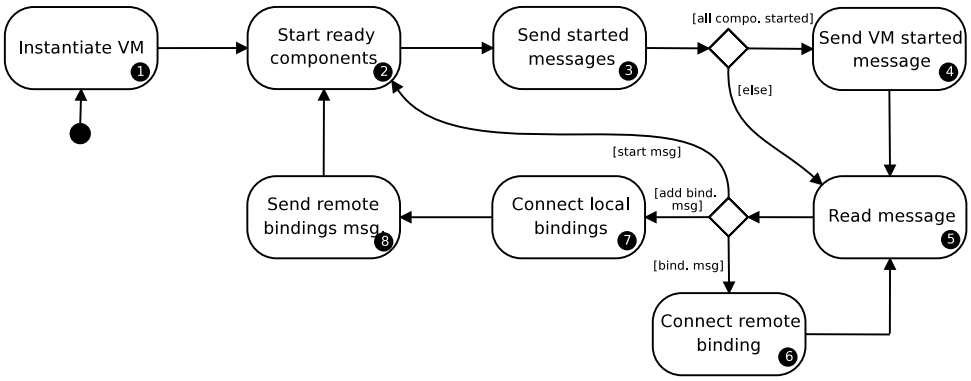

Figure 6: VM Manager Activity Diagram: Up Phase

Unbinding and shutdown. A VM manager is in charge of stopping some local components, or all its components when the VM is to be destroyed

(Figure 7, ❶), *i.e.*, removed from the running application. In this case, all the components hosted on that VM need to be stopped and all bindings on these components (connected to imports or exports) need to be removed. If a component involved in the shutdown process does not provide any service (there is no component connected to it), it can immediately stop, and all outgoing bindings can be removed for these components (❷). Otherwise, it cannot stop before all components connected to it on mandatory imports have unbound themselves. To do so, the VM manager of the VM under destruction first sends *unbind required* messages to all VMs hosting components connected to those VM's components (❸). The VM manager of the VM to be destroyed then collects *unbind confirmed* messages (❹) and stops the corresponding components when all components using that component on mandatory imports have stopped and unbound (❺). Whenever a component stops, an *unbind confirmed* message is sent (❻). The VM is destroyed and the CM informed when all components are stopped (❼).

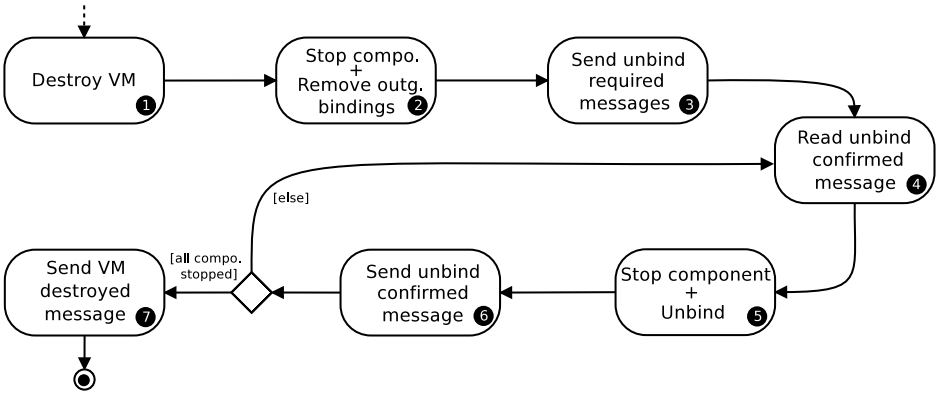

Figure 7: VM Manager Activity Diagram: Down Phase (Destruction)

As a side effect to a VM destruction, the other VM managers can receive messages (Figure 8, ❶) from their partner VMs. Upon reception of an *unbind required* message, the VM manager either stops and unbinds some components (❷), if possible (no bindings on them or bindings on remote optional imports only), or sends similar messages for all remote components bound on mandatory imports to its components (❺). When a VM manager stops (and unbinds) a component (❷), it may send a message to the CM indicating that the VM is not fully operational (❸). It also sends messages to all remote partner components formerly providing a service to that component, to

let them know that this component has been stopped/unbound (❹). Upon reception of an *unbind confirmed* message, the VM manager goes to step ❷.

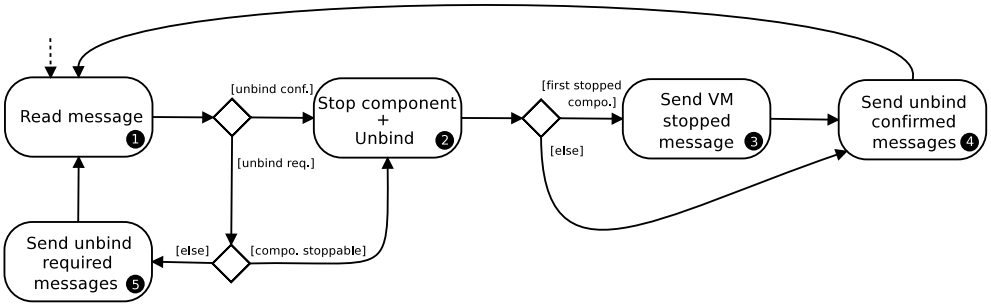

Figure 8: VM Manager Activity Diagram: Down Phase (Side Effect)

Components bound on optional imports just need to unbind themselves, but do not need to stop. Local bindings are handled locally by the VM manager, but these changes can impact other remote components, and in that case additional *unbind required* messages may be emitted. The component shutdown implies a backward propagation of *unbind required* messages and, when this first propagation ends (on components without exports or with optional imports only), a second forward propagation of *unbind confirmed* messages starts to let the components know that the disconnection has been actually achieved. These propagations terminate because there is no cycle of bindings over mandatory imports.

2.6. Failure Recovery

Many different kinds of infrastructure failures may affect the normal operation of cloud applications, going from those due to human errors or spikes on customer demands to security breaches. Any of the reconfiguration operations considered here may fail due to many different reasons, from a hardware failure to a credentials failure. From all these possible failures, we may abstract from the reasons and focus on the consequences. In this work we focus on VM failure.

We assume that a VM may fail at any time, and that eventually the CM detects such a failure (by polling, heartbeat, or some other method) and reacts to lead the application under its control to a consistent state. Specifically, when a failure is detected, the CM performs the following tasks:

- The CM first updates the model of the active system by removing the failed VM;

- It also purges its buffers, removing all messages coming from or addressed to the failed VM;
- Finally, the CM alerts the impacted VMs (connected to the failed VM) of the failure sending *failure alert* messages.

Upon the reception of a *failure alert* message reporting the failure of a neighbor VM, a VM manager performs the following tasks:

- It purges its buffers;
- It changes the current states of its local components by unbinding and stopping the impacted components; and
- It sends *shutdown component* messages to all VM managers of VMs containing components connected to its shutdown components.

When a VM manager receives a *shutdown component* message resulting of the failure propagation, it stops the impacted local component and propagates (locally and remotely) this shutdown. Notice that the propagation of *shutdown component* messages goes in one direction, and since only active components may be turned off, the process eventually terminates. It is worth noting that multiple VM failures may occur, this can be due to failures of different instances of a single VM or failures of different VMs. A failure can also take place when a VM is already handling a failure involving another VM (cascading failures). If the number of failures is finite and if there is no cycle of bindings through mandatory imports, the reconfiguration protocol always succeed in restoring the application consistency. In the worst case, all components are stopped. In Sections 3.2 and 3.3, we will show how this process has been verified, and how after an instability period in which certain invariants may be violated, the system eventually reaches a state in which they are again satisfied.

2.7. Examples of Reconfiguration Scenarios

We show in this section how the protocol works on simple reconfiguration scenarios for the Web application presented in Figure 1. Let us assume that the application is fully operational and all components on all VMs are started (end of the first *up* phase in Figure 5). A new version of the MySQL database management system is available and we decide to upgrade that component to this new version. Accordingly, the cloud manager initiates a *down* phase

(middle part of Figure 5) characterized by an emission of a *remove* message to VM3. We show in Figure 9 a Message Sequence Chart (MSC) overviewing the interactions and behaviors of all participants (CM and VM managers) for this specific scenario. It is worth noting that we do not focus on the data storage and migration here (not the goal of the protocol), which should be handled by the designer if needed.

Upon reception of the *remove component* message, VM3 sends an *unbind required* message to VM2 requesting to unbind the Tomcat component from the MySQL component. When VM2 receives this message, it cannot unbind immediately because Tomcat is used by a remote component (Apache), therefore it sends too an *unbind required* message to VM1. Upon reception of that message, the VM1 manager stops the Apache component, because no other component is connected to it, and then unbinds the Apache component from the Tomcat component. VM1 sends a confirmation message to VM2 indicating that the disconnection has been achieved. VM1 also sends a *VM stopped* message to the CM indicating that its components are not started anymore. When VM2 receives the *unbind confirmed* message, its manager stops Tomcat and unbinds it from MySQL. A confirmation is sent from VM2 to VM3 and a *VM stopped* message is sent to the CM. Once VM3 receives the confirmation message, its manager stops the MySQL component, and sends an acknowledgement message to the CM indicating that the VM is stopped too. Note that stopping Tomcat and Apache is required to preserve architectural invariants: a started component cannot be connected to a stop component.

After the removal of MySQL, the application is in a situation where components Apache and Tomcat are off and components Profiling and Object Cache are on.

In order to restore a fully operational application, let us now consider an *up* scenario (right-hand side of Figure 5) where the CM manager adds a new version of the MySQL component on VM3 (*add* message) and a new binding between the Tomcat component and the new MySQL component. We show in Figure 10 the interactions and actions involved in this scenario. VM3 can start the MySQL' component immediately because this component does not require any service from other components (no imports). VM3 knows that VM2 needs to connect its component to the MySQL' component, therefore the VM3 manager posts a *send export* message with the connection information to VM2. Upon reception, the VM2 manager can connect both components. The VM3 manager also indicates to VM2 that its MySQL'

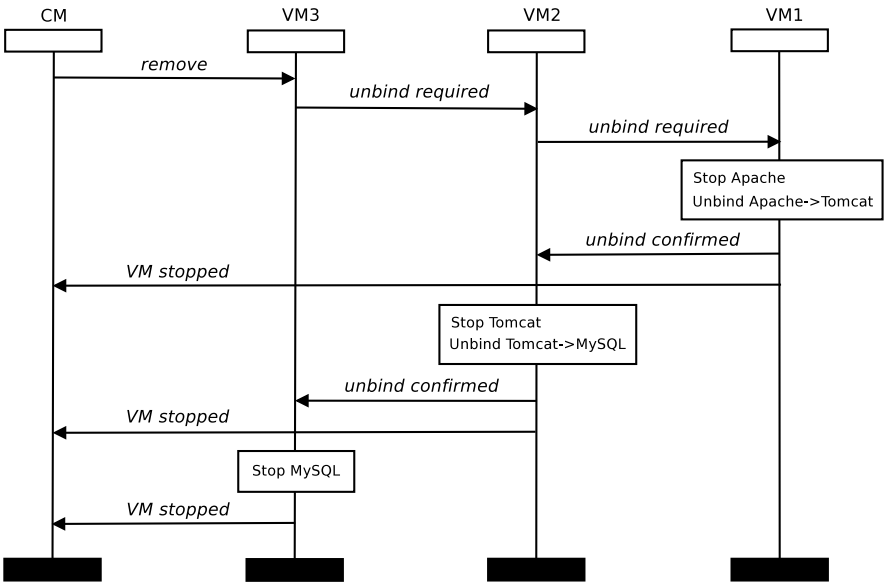

Figure 9: MySQL Removal Scenario

component has started and to the CM that VM3 is started. Upon reception of the *send export* message, the VM2 manager starts the Tomcat component. VM2 sends a *send export* message and a *started* message to VM1, because the VM2 manager knows the dependency between the Apache component and the Tomcat component. VM2 also informs the CM that VM2 is started. The VM1 manager finally binds Apache to Tomcat, starts the Apache component, and informs the CM that VM1 is started too. Thus, the system is back to operational and all components are active again. Note that acknowledgement messages are not systematically required. They are useful in some specific cases, *e.g.*, when a component (import side) expects its partner (export side) to start.

Finally, imagine that a failure of VM3 occurs. Figure 11 details this scenario. First the CM detects this failure, updates the current model of the application, purges its buffers, and sends a *failure alert* message to VM2. Upon reception of this message, the VM2 manager purges its buffers and restores its local consistency by stopping and unbinding the Tomcat component. Then, it sends a *shutdown component* message to VM1. Upon reception of this message, the VM1 manager stops and unbinds the Apache component. Both VM managers also send messages to the CM to let it know that they

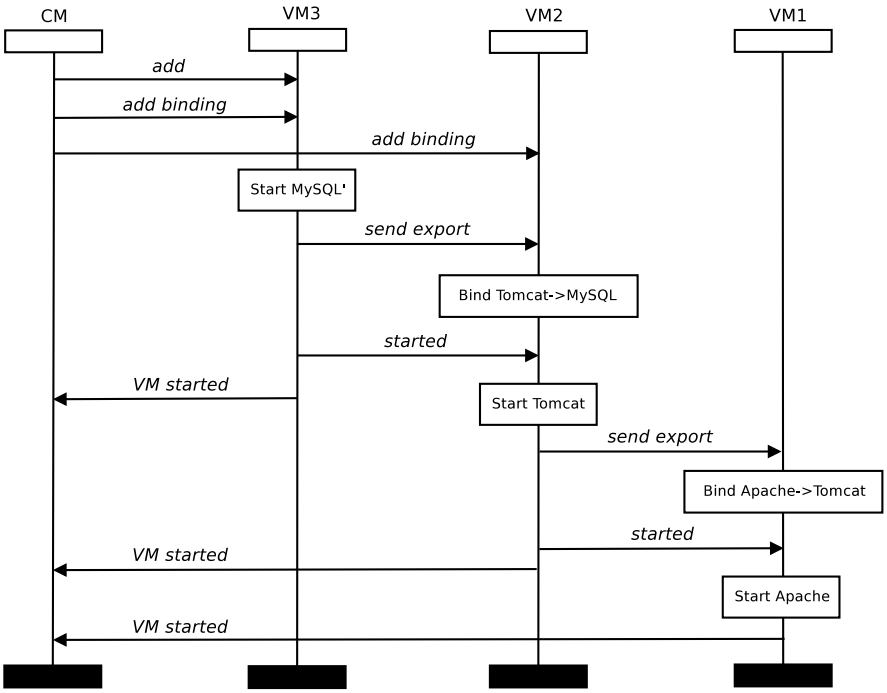

Figure 10: MySQL' Addition Scenario

are no longer started.

3. Rapid Prototyping and Verification with Maude

We chose Maude [10] for implementing the reconfiguration protocol because its declarative style facilitates program writing, and specifically, it is very simple to specify the creation and destruction of objects, and to model locality as objects that contain other objects. Moreover, Maude is adequate to specify concurrent systems and is equipped with a large variety of analysis tools. All sources for our Maude implementation and its verification are available online (see [1]).

3.1. Maude Specification

In this section, we describe the Maude representation of the models and present some of the rewrite rules so that the reader can gain a general understanding of how the different steps of the protocol are represented as rules. Maude supports the modeling of object-based systems by providing

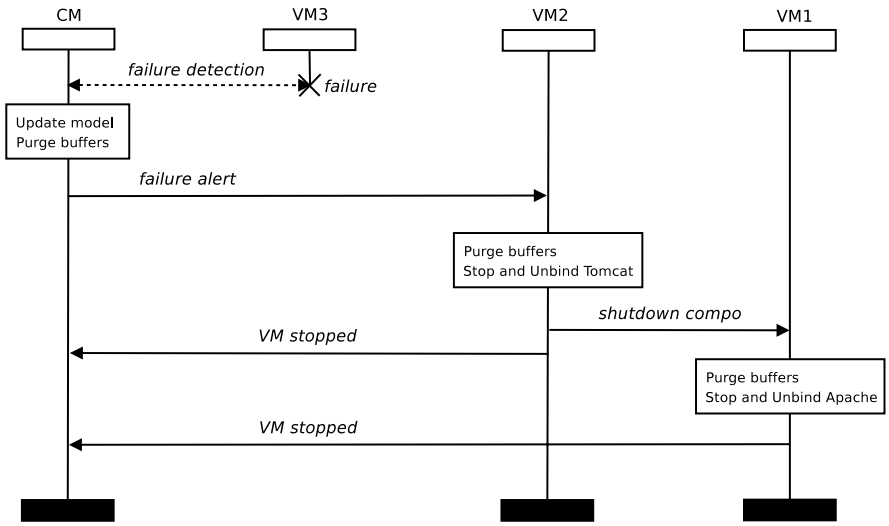

Figure 11: Failure of VM3

sorts representing the essential concepts of object (`Object`), object identifier (`Oid`), message (`Msg`), and configuration (`Configuration`). A configuration is a multiset of objects and messages (with the empty syntax, associative commutative, union operator __) that represents a possible system state.

Given a class C with attributes a_i, of respective types S_i, the objects of this class are then record-like structures of the form `<` O `:` C `|` a_1`:`v_1`,` ..., a_n`:`v_n `>`, where O is the identifier of the object, and v_i are the current values of its attributes. Objects may interact via messages, which are terms of sort `Msg`. We assume all messages are of the form `to` O `:` MB, where O is the addressee of the message and MB its message body, a term of sort `MsgBody`.

For the specification of the protocol, we represent a system as an object of class `CloudManager` and a collection of objects of class `VMManager`. `CloudManager` and `VMManager` objects communicate through asynchronous messages. We abstract such a common behavior in `BufferedClass`, a class with two attributes, `in` and `out`, both of type `Queue{Msg}`, representing the in and out message buffers, respectively.

```
class BufferedClass | in: Queue{Msg}, out: Queue{Msg} .
```

Given variables `O1` and `O2` of sort `Oid`, `V1` and `V2` of sort `BufferedClass`, `In` and `Out` of sort `Queue{Msg}`, and `MB` of sort `MsgBody`, the transfer of a message

from the output buffer of an instance of the `BufferedClass` class to the input buffer of another one is modeled by the following `MsgTransfer` rule:

```
rl [MsgTransfer] :
   < O1: V1 | out: (Out; to O2: MB) >
   < O2: V2 | in: In >
=> < O1: V1 | out: Out >
   < O2: V2 | in: (to O2: MB; In) > .
```

Both `CloudManager` and `VMManager` classes are declared as subclasses of `BufferedClass`, thus inheriting attributes `in` and `out`. The above `MsgTransfer` rule, dealing with the transmission of messages, is also applicable on them, so that they inherit such a behavior. The `CloudManager` class has three additional attributes: `actions`, which is used to represent the scenario or sequence of reconfiguration operations to be executed, `VMs`, a map that associates a Boolean value to a virtual machine indicating whether it is started or not (the map returns `null` for identifiers not in the map), and `phase`, the phase (up or down) of the overall system.

```
class CloudManager |
      actions: List{Msg},  phase: Phase,
      VMs: Map{Oid, Bool}, Cs: Map{Tuple{VMId, CId}, State} .
subclass CloudManager < BufferedClass .
```

To illustrate the form in which a CM obtains the sequence of actions, let us consider the Web application example shown in Figure 1. The sequence of actions corresponding to the second part of the scenario presented in Section 2.4 is: `remove(MySQL); add(MySQL'); addBinding(Bd')`, with `Bd'` a set of bindings. This scenario is interpreted as `down; remove(MySQL); up; add(MySQL'); addBinding(Bd')` by the cloud manager, which infers the necessity of two phases for applying this scenario.

The `VMManager` class has a single attribute, `cs`, in which a `VMManager` object keeps all the information required on the components, their ports and their bindings as a collection of objects.

```
class VMManager | cs: Configuration .
subclass VMManager < BufferedClass .
```

The `cs` attribute keeps a collection of components, where a component is represented as an instance of class `Component`, which keeps in its attributes an import set (`imps`), an export set (`exps`), and its activation state (`started`).

```
class Component | started: Bool, imps: Set{Port}, exps: Set{Port} .
```

A port has an identifier, and can either be an import or an export. Imports and exports are constructed by using the following operators:

```
sorts Port Ploc CImp .
ops optional mandatory : -> CImp .
op _._._ : Oid Oid Oid -> Ploc .
op imp : Oid CImp Maybe{PLoc} Bool -> Port .
op exp : Oid Map{PLoc, Bool} -> Port .
```

Sort `CImp`, of importation types, corresponds to the optional or mandatory import property. Terms of sort `PLoc`, of port locations, are constructed as triples *VMId* . *CId* . *PId*, with *VMId* a VM object identifier, *CId* a component object identifier, and *PId* a port identifier. The first argument of constructors `imp` and `exp` represents the identifier of the corresponding port. An import is only connected to one export, or is not connected. The second argument of the `imp` operator represents the export it is bound to. The `Maybe` parametric sort can either be an element of its parameter, a port location in the case of `Map{PLoc}`, or `null`. Thus, if the port is bound to an export, it will keep the location, or `null` otherwise. The third argument of the `imp` operator indicates whether the component it is connected to is active or not. An export can be connected to zero or more imports. For each import connected to an export, the `exp` operator allows us to keep its location and a Boolean value indicating whether the remote component is active or not.

Figure 12 shows the Maude term for the Web Application Model in Figure 1. We can see a CM, three VMs with their respective components, and how given names for VMs, components and ports, bindings are represented as appropriate references in the ports of the components.

Once the different classes and data structures are defined, as well as the operations for handling them, the reconfiguration algorithm itself is defined using rewrite rules. To do so, we defined 14 rules for the start-up process, 29 rules for the shutdown process, and 3 rules for the VM failure recovery. These rewrite rules implement the different actions that can take place in the algorithms, *e.g.*, for the start-up process: if a cloud manager gets an action, such as, *e.g.*, `instantiateVM`, it processes it by creating a new `VMManager` object; if a component is ready to be started and all its mandatory imports are satisfied, it becomes active; when a VM is being instantiated and all its components are active, it finalizes its instantiation and sends an acknowledgement message to the cloud manager; upon reception of an instantiation acknowledgement message from a VM manager, the cloud manager updates its `VMs` attribute; etc. We explain several of these rules below.

```
< DEP: CloudManager |
   in: nil,
   out: nil,
   actions: nil,
   VMs: ((VM1, true), (VM2, true), (VM3, true)),
   Cs: (((VM1, Apache), true), ((VM1, Profiling), true),
        ((VM2, Tomcat), true), ((VM2, Cache), true),
        ((VM3, MySQL), true)),
   phase: up >
< VM1: VMManager |
   in: nil,
   out: nil,
   cs: (< Apache: Component |
             imps: (imp(AI1, optional, VM1.Profiling.PE, true),
                    imp(AI2, mandatory, VM2.Tomcat.TE, true)),
             exps: empty,
             started: true >
         < Profiling: Component |
             imps: empty,
             exps: exp(PE, (VM1.Apache.AI1, true)),
             started: true >) >
< VM2: VMManager |
   in: nil,
   out: nil,
   cs: (< Tomcat: Component |
             imps: (imp(TI1, mandatory, VM2.Cache.CE, true),
                    imp(TI2, mandatory, VM3.MySQL.ME, true)),
             exps: exp(TE, (VM1.Apache.AI2, true)),
             started: true >
         < Cache: Component |
             imps: empty,
             exps: exp(CE, (VM2.Tomcat.TI1, true)),
             started: true >) >
< VM3: VMManager |
   in: nil,
   out: nil,
   cs: < MySQL: Component |
            imps: empty,
            exps: exp(ME, (VM2.Tomcat.TI2, true)),
            started: true > >
```

Figure 12: Maude Term for the Web Application Model in Figure 1

The `ComponentActivation` rule (Figure 13) corresponds to the activation of a component when that component is *startable*—the `isStartable` auxiliary function checks whether all mandatory imports are satisfied. In this case, the `started` attribute is changed to `true`, and the components connected to the one being activated are updated—by direct modifications on components in the same VM with the `updateLocalActiveBoolean` auxiliary function, or by sending appropriate messages to components in different VMs (the `activationMsgs` function generates these messages). To avoid wrong activations, when a VM is initially instantiated, one token per component (`startToken`) is added to the VM inner configuration. When a component is started, its token is removed.

```
crl [componentActivation] :
   < VMId : VMManager |
       cs: (< CId: Component |
                 imps: Imps, exps: Exps, started: false >
            startToken(CId)
            Cs),
       out: Out >
=> < VMId: VMManager |
       cs: updateLocalActiveBoolean(CId,
             < CId: Component | imps: Imps,  exps: Exps, started: true >
             Cs),
       out: (activationMsgs(VMId, CId, Imps, Exps)) Out) >
if isStartable(Imps) .
```

Figure 13: Component Activation Rule

Rules `addBindingsAction` and `addExportBindingsMsg` in Figure 14 are in charge of establishing the bindings between components (Figure 14). When a VM manager receives an `addBindigs` message, it resolves all local binding requests and generates the necessary messages for remote connections (`addBindings(VMId, Cs, Bds, nil)`). In rule `addExportBindingsMsg`, components with `bindingRequest` messages update their imports with the information on the connection. In case the component is active, it sends an `activation` message to the component at the other side of the connection so that it can update its connection information.

The rule `destroyVMReceive` corresponds to the consumption of a destruction message by a VM manager (Figure 15). The manager generates local tokens indicating that the VM and all the components on that VM must stop.

```
rl [addBindingsAction] :
   < VMId: VMManager | cs: Cs,
     in: (In to VMId: addBindings(Bds)),
     out: Out,
     Atts1 >
=> < VMId: VMManager | cs: getConf(addBindings(VMId, Cs, Bds, nil)),
     in: In,
     out: (getMsgs(addBindings(VMId, Cs, Bds, nil)) Out),
     Atts1 > .
```

```
rl [addExportBindingsMsg] :
   < VMId: VMManager |
     cs: (< CId: Component |
            imps: (imp(PId, CI, null, false), Imps),
            started: B1,
            Atts2 >
          Cs),
     in: (In
         to VMId: bindingRequest(VMId.CId.PId, VMId'.CId'.PId', B)),
     out: Out,
     Atts1 >
=> < VMId: VMManager |
     cs: (< CId: Component |
            imps: (imp(PId, CI, VMId'.CId'.PId', B), Imps),
            started: B1,
            Atts2 >
          Cs),
     in: In,
     out: (if B1
           then to VMId': activation(CId', PId', VMId.CId.PId)
                Out
           else Out
           fi),
     Atts1 > .
```

Figure 14: Rules Establishing the Bindings between Components

The manager also posts messages to all VMs hosting components connected to its components, indicating that they must unbind them.

The occurrence of VM failures is modeled with a `failure` rule, which can be applied at any time on any VM. To limit its application, there is a failure-injector class whose instances are in charge of limiting the occurrence of such failures (`nbf` and `cnbf` represent the number of failures to be injected

```
rl [destroyVMReceive] :
   < VMId: VMManager |
     cs: Cs,
     out: Out,
     in: In to VMId: destroy,
     Atts >
=> < VMId: VMManager |
     cs: (tokenDestroy(VMId)
          generateTokenStopCs(Cs)
          Cs),
     in: In,
     out: generateUnbindMsg(VMId, Cs) Out,
     Atts > .
```

Figure 15: Destruction Message Reception Rule

and the number of failures already injected, respectively).

```
class FailureInjector | nbf: Int, cnbf: Int .
```

During the protocol execution, the `failure` rule applies the number of times specified by the `nbf` attribute and can be fired at any time. This is particularly convenient for validation purposes using model checking techniques, because this provides an exhaustive enumeration and analysis of all possible execution cases.

3.2. Simulation and Model Checking

Simulation is very useful for exploring an execution of a system, with the possibility of using some strategy language to guide such execution. In Maude, simulation relies on rewriting, which consists in successively applying equations and rewrite rules on an initial term (an application model here). Since systems may be rewritten in many different ways, Maude also provides a searching command, which allows us to explore the reachable state space up to certain depth. Thus, we may perform analysis on the reachability of states satisfying certain conditions, *e.g.*, for deadlock states or for final states with non-consumed messages. It is useful for checking non-trivial situations, for instance, we were able to check in this way that up and down phases are confluent, in the sense that for each of the models used we reach a unique final model, which is moreover the expected one.

Simulation and reachability analysis were very helpful for identifying and fixing simple bugs in the first versions of the protocol. Beyond these basic

checks, we used Maude's Linear Temporal Logic (LTL) explicit-state model checker [12] for analyzing all possible executions given an application model and a reconfiguration scenario. Maude's model checker allows us to check whether every possible behavior starting from a given initial model satisfies a given LTL property. It can be used to check safety and liveness properties of rewriting systems when the set of states reachable from an initial state is finite. Full verification of invariants in infinite-state systems can be accomplished by verifying them on finite-state abstractions [22] of the original infinite-state system, that is, on an appropriate quotient of the original system whose set of reachable states is finite.

We identified several properties that the protocol must respect during any step of its application. They helped to verify that key architectural invariants are satisfied during the protocol execution (*e.g.*, P1, P5, or P8 below), final objectives are fulfilled (*e.g.*, P2, P3, or P6), and some intermediate behavioral constraints respected (*e.g.*, P4 or P7). The main ones may be summarized as follows:

P1 All mandatory imports of a started component are bound to started components.

P2 All VMs are eventually started (resp., all components started).

P3 A VM being destroyed eventually succeeds in stopping all its components.

P4 All reconfiguration operations submitted by the CM are always treated by the recipient VMs.

P5 There is never a binding from a started component (import side) to a stopped component (export side).

P6 All bindings to be added/removed are eventually added/removed.

P7 Each `startToken` message is eventually consumed. Similar properties are stated for other messages.

P8 If there is a failure, then eventually a state is reached where architectural invariants are re-established.

Note that some properties depend on the input application model / reconfiguration scenario, *e.g.*, we can imagine specific scenarios in which P2 is not verified because a binding is missing for a mandatory import.

We now illustrate how such properties can be specified in LTL by using state predicates. In order to check P1, for instance, we may define a state

predicate, `satisfied-imports`, which is true if all components have their imports satisfied, and false otherwise. In this case we may define it by using matching equations: if there is a port `VMId.CId.PId` failing the condition, then it is false for the entire system, and it is true otherwise. Given variables `VMId`, `CId` and `PId` of type `Oid`, `PLoc?` of type `Maybe{PLoc}`, `Imps` of type `Set{Port}`, and `Cs` and `Conf` of type `Configuration`, the satisfaction of the proposition may be defined as shown in Figure 16.

```
op satisfied-imports : -> Prop [ctor] .
eq { < VMId: VMManager |
       cs: (< CId: Component |
              imps: (imp(PId, mandatory, PLoc?, false), Imps),
              started: true >
           Cs) >
     Conf }
   |= satisfied-imports
  = false .
eq { Conf } |= satisfied-imports = true [owise] .
```

Figure 16: Specification of the `satisfied-imports` predicate

State predicates are defined as operators of sort `Prop`, and their semantics are given by means of equations. Notice that the `imp` operator takes as third argument a value of type `Maybe{PLoc}`, meaning that the argument also accepts `null` as value. Thus, the first equation above matches a system if there is a component with its `started` attribute `true` and with a mandatory import, unbound or bound to a non-active component. If there is no such component in the system the second equation—the `owise` attribute makes of it an otherwise equation—is applied, returning `true` when the operator is evaluated. Given the above definition of this proposition, we can now check an LTL formula like `[] satisfied-imports`.

3.3. Experimental Results

The reconfiguration algorithm has been validated on more than 300 examples (application model and reconfiguration scenario), representing typical n-tier Web applications with different sizes and connection structures. Most of these case studies were extracted from real cases, others present artificial configurations covering all possible combinations (please, see [1] for a detailed presentation of these results). Table 1 shows some experimental results for

the verification of properties P1-P8 (for P8, we show the results for one and two failures), on several application models from our database. For each application model (identified in the first column), the verification times (given in seconds) of several reconfiguration operation sequences ('ops' column) are given for each of these eight properties (last nine columns).[4] Note that we give the times to complete the verification, that is, to explore the entire state space. Simulation times, following a single path of execution, are under 5 milliseconds in all cases. The size of the state space depends both on the size of the application and on its complexity.

Table 1: Experimental results for properties P1-P8 on several examples

Id	Size				Analysis time (secs)								
	VMs	Cs / lbds	rbds	ops	P1	P2	P3	P4	P5	P6	P7	P8(1)	P8(2)
001	3	(1-1-0) (0-0-1) / 1 (2-0-1) (0-0-1) / 1 (0-0-1) / 0	2	4/2	< 1	$< 1^*$	< 1	< 1	1	2	< 1	4	5
				4/1/2	< 1	< 1	< 1	< 1	1	2	< 1	4	5
				4/3/4	2	2	2	2	2	4	2	10	13
002	4	(3-0-0) / 0 (2-0-2) / 0 (0-0-1) / 0 (0-0-1) (0-1-1) / 0	6	5	82	91	92	93	103	188	103	1105	2394
				5/1	110	111	110	111	117	210	107	1260	5219
				5/2	111	111	110	111	117	210	107	2255	5304
				5/1/2	111	111	111	111	117	209	107	2590	6285
003	5	(2-0-0) / 0 (1-0-1) / 0 (1-0-2) (1-0-1) / 1 (1-0-1) / 0 (0-0-1) / 0	5	6/2	23	23^*	24	24	29	82	25	500	1687
				6/1/2	23	23^*	24	25	29	82	25	505	2301
				6/2/3	29	25^*	28	28	32	88	28	949	4363
				6/3/4	62	26^*	64	65	74	152	71	5032	$> 3h$

The size of a model is described by its number of virtual machines (VMs), number of components, ports, and local bindings in each of these VMs (Cs / lbds, with each line describing the components and local bindings of each VM), and number of remote bindings (rbds). A component is represented by a triple (*MI*-*OI*-*E*), where *MI*, *OI* and *E* are its numbers of mandatory imports, optional imports, and exports, respectively. Operation sequences are given just by the number of reconfiguration operations in each successive phase separated by slashes. A sequence 4/1/2 represents a sequence of 4 operations in an up phase (typically instantiations for each of the three VMs plus an operation for adding the bindings), followed by 1 operation in a

[4]Although Maude gives times in milliseconds, to improve readability, all values were rounded to seconds. A time < 1 indicates that the time taken by the analysis was smaller than one second.

down phase (a VM destruction), and 2 more operations in a final up phase (instantiation of a new VM and bindings addition).

The complexity of the applications, understood as how intricate are the bindings between mandatory/optional imports and exports, is not shown in the table, although it is clearly reflected in the numbers. Models 002 and 003 are very similar in size, but very different in their bindings. Although 003 has a non-trivial chaining of mandatory imports, 002 has a cycle in its bindings (with optional ports), which explains that although 003 is slightly larger, the analysis for 002 takes more than double for simpler operation sequences. The complexity of the up/down operations in distinct scenarios may be very different. For instance, the destruction of a specific VM may induce many (shutdown) operations by propagation, whereas the destruction of another VM, *e.g.*, a VM hosting components without any exports, will not generate any additional operations. The numbers marked with a star indicate that the analysis fails and returns a counterexample. As we pointed out above, P2 is not valid for all sequences of operations.

We may observe from the results in the table that for some properties the numbers are very similar. For each model-checking command, Maude constructs a Büchi automaton from the negation of the property formula and lazily searches the synchronous product of the Büchi automaton and the system state transition diagram for a reachable accepting cycle. Since the state spaces are exhaustively explored, the properties are the only difference, but for such properties the products are very similar in nature. The LTL formula for P8 is more complex than the others, and the state spaces much bigger.

Last but not least, let us note that the analysis times show that verification is not (yet) a zero-cost task. Nonetheless, this is worth it if software quality is at stake and this is the case with the reconfiguration protocol in this paper. Most issues and bugs are usually found on models involving a few VMs, thus the execution times show that such models can be analyzed in a reasonable time (within one hour). The execution times are also useful to show how the input (architecture and scenario) impacts the verification time, and thus gives an idea of how these analysis tasks scale. There are other approaches where verification techniques have been used for verifying cloud management protocols or applications, see, *e.g.*, [2, 5, 14]. These related works present experimental results very similar to ours in terms of analysis time.

3.4. Found Issues

Beyond correcting very simple bugs identified using simulation and reachability analysis, model checking techniques helped us to detect very subtle bugs in pathological models that would have been very difficult to identify without such exhaustive validation techniques. All detected bugs were corrected in the final implementation of the protocol. Let us focus on two concrete issues.

First, in an intermediate version of the protocol, we did not have the notion of up/down phases, and it was possible to instantiate VMs and destroy others at the same time. Nonetheless, this typically generates spurious messages and erroneous behaviors, *e.g.*, a VM trying to start and connect its components to the components of another VM, which is in the process of destroying itself. These contradictory instructions exchanged during the protocol execution motivated our decision to introduce up and down phases. This simplified the protocol execution and allowed us to ensure that the protocol preserves the consistency of the application at any step of its execution.

Second, when destroying VMs, we realized that the components involved in this process were not stopped properly. This was detected thanks to property P1. We detected that the propagation was not achieved as expected in some boundary cases, *e.g.*, for combination of local and remote bindings on mandatory imports. In that situation, the first backward propagation of *unbind required* messages stopped when arriving on local bindings. This was a bug, because local bindings can lead to other remote bindings that require proper disconnection too. On a wider scale, the double propagation, which was introduced to correctly stop components, required several corrections and adjustments before working properly and respecting all the properties mentioned earlier in this section.

4. Related Work

Dynamic reconfiguration is not a new topic and many solutions have already been proposed in the context of, *e.g.*, software architectures [18, 21, 17, 4, 19], graph transformation [3, 28], software adaptation [25, 24], metamodelling [16, 20], or reconfiguration patterns [8]. In software architectures, for example, the authors proposed various formal models, such as Darwin [18] or Wright [4], in order to specify dynamic reconfiguration of component-based systems whose architectures can evolve (adding or removing components and connections) at run-time. These techniques aim at helping users to formally

design dynamic applications. In [17, 19], the authors show how to formally describe behavioral models of components using FSP and analyze these models using the Labeled Transition System Analyser (LTSA), which allows the verification of temporal properties on the component architecture. Our goal here is to propose a reconfiguration protocol that automatically applies a set of reconfiguration tasks on a component assembly distributed on several VMs. We do not want to check properties on each component assembly, we rather want our protocol itself to preserve some important properties during its application, whatever the assembly being reconfigured.

In the cloud computing area, there are several configuration management tools, such as Puppet or Chef. They allow to automatically provision and configure new machines as described in configuration files called manifests or recipes. Such tools do not provide advanced protocols, which ensure to preserve consistency of the application being (re)configured. Some existing environments also provide mechanisms to automatically scale deployed applications based on monitoring data (see, *e.g.*, the Elastic Beanstalk from Amazon Web Services). However, these approaches typically work at the application level (Platform-as-a-Service, PaaS). Moreover, changes are triggered with respect to the individual performance of each tier, although there are attempts to decide elasticity actions from entire application performance models, see, *e.g.*, the Reservoir [9] or ConPaaS projects [23]. We do not address here the question of when or why new copies of components or new VMs are needed, or when existing ones can be disposed of. We assume that a running application required some reconfigurations and we present the mechanisms to apply these changes.

In [13, 27], the authors present a protocol that automates the configuration of distributed applications in cloud environments in a decentralized way. Each VM is in charge of starting its own components and to do so needs to interact with the other VMs in order to exchange binding information. In these applications, all elements are known from the beginning (*e.g.,* numbers of VMs and components, bindings among components, etc.). This approach works fine when the application does not need to be changed after deployment. Unfortunately, this is not the case in the cloud, where most applications need to be reconfigured for considering new requirements, scaling on-demand, or applying failure recovery techniques. Another recent related work [15] presents a system that manages application stack configuration. It provides techniques to configure services across machines according to their dependencies, to deploy components, and to manage the life cycle of installed

resources. This work presents some similarities with ours, but [15] does not focus on composition consistency, architectural invariants preservation, or robustness of the reconfiguration protocol.

As far as reconfiguration of component assemblies is concerned, [7, 6] present a reconfiguration protocol applying changes to a set of connected components for transforming a current assembly to a target one given as input. Reconfigurations steps aim at (dis)connecting ports and changing component states. The protocol is robust in the sense that all the steps of this protocol preserve a number of architectural invariants. For designing this reconfiguration protocol, the authors used value-passing process algebra and model checking techniques for detecting and correcting behavioral issues that showed up during the protocol design [7]. They proved recently the protocol correctness by using theorem proving techniques [6]. This protocol does not easily scale to cloud applications because the authors assume that all components are hosted on a same VM and a unique centralized manager is in charge of the reconfiguration steps. Our protocol instead is fully concurrent (all VMs evolve independently from one another at different speeds). In [2], the authors present a formally validated management protocol for instantiating and removing VMs from a running cloud application. The protocol is quite different because in that case, bindings are not made explicit and the main task of the protocol is to resolve port matching by using port typing and a publish-subscribe messaging system. Note also that Maude has already been used for analyzing cloud architectures and applications, see, *e.g.*, [29, 5].

5. Conclusion

In this paper, we have presented a new protocol for automatically reconfiguring cloud applications consisting of interconnected components distributed over several VMs. The protocol does not only support VM instantiation and component start-up, but also VM destruction and component shutdown. These management tasks are guided by reconfiguration operations posted through a cloud manager. All VMs work in a decentralized and loosely-coupled way in order to apply these reconfiguration tasks, exchanging messages when necessary via FIFO buffers. The protocol is robust in the sense that it preserves composition consistency and well-formedness architectural invariants at any step of its application. The protocol is reliable because it detects VM failures and makes the application restore a global consistent

state. For ensuring these correctness requirements, we have implemented it using Maude's rewriting logic-based language. This results in a formal description of the protocol that we analyzed using Maude's verification tools for chasing subtle bugs in pathological cases and therefore ensuring that our implementation satisfies some key properties and invariants. As a result, this allowed us to identify and correct several subtle issues during the protocol's design. It is worth noting that a Java implementation of the protocol is under development at Orange Labs in the context of the OpenCloudware funded project.[5]

A first perspective of this work aims at getting rid of the alternative up and down phases during the protocol application. This is a non-trivial change, since start and stop messages may be unwillingly mixed up, and each VM manager must handle them correctly without introducing inconsistencies in the application being modified. Another perspective regards high-level properties that may be preserved by the application. Our reconfiguration protocol can be viewed as the (low-level) mechanics to achieve a set of reconfigurations, but our protocol is not "clever" in the sense that there is no algorithm on top of it computing the correct sequences of reconfigurations to ensure high-level properties such as availability or elasticity. Such algorithms could be proposed in order to satisfy these additional requirements. As far as validation aspects are concerned, we plan to formally prove the protocol correctness using Maude's invariant analyzer tool (InvA). Another alternative in this direction is to rely on Hoare-style verification and to use graph-like structures [26]. Finally, we would like to use Maude's statistical model checker to analyze the protocol. Probabilistic modeling and analysis is necessary to handle non-deterministic events such as network delays, congestion, or failures.

Acknowledgements. This work has been partially supported by the SeaClouds EU project and the OpenCloudware project, which is funded by the French *Fonds national pour la Société Numérique* (FSN), by *Pôles* Minalogic, Systematic, and SCS.

References

[1] `http://maude.lcc.uma.es/HRfCA`.

[5] `http://opencloudware.org`

[2] R. Abid, G. Salaün, F. Bongiovanni, and N. De Palma. Verification of a Dynamic Management Protocol for Cloud Applications. In *Proc. of ATVA'13*, volume 8172 of *LNCS*, pages 178–192. Springer, 2013.

[3] N. Aguirre and T. Maibaum. A Logical Basis for the Specification of Reconfigurable Component-Based Systems. In *Proc. of FASE'03*, volume 2621 of *LNCS*, pages 37–51. Springer, 2003.

[4] R. Allen, R. Douence, and D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Proc. of FASE'98*, volume 1382 of *LNCS*, pages 21–37. Springer, 1998.

[5] L. Bentea and P. Csaba Ölveczky. A Probabilistic Strategy Language for Probabilistic Rewrite Theories and its Application to Cloud Computing. In *Proc. of WADT'12*, volume 7841 of *LNCS*, pages 77–94. Springer, 2013.

[6] F. Boyer, O. Gruber, and D. Pous. Robust Reconfigurations of Component Assemblies. In *Proc. of ICSE'13*, pages 13–22. IEEE/ACM, 2013.

[7] F. Boyer, O. Gruber, and G. Salaün. Specifying and Verifying the Synergy Reconfiguration Protocol with LOTOS NT and CADP. In *Proc. of FM'11*, volume 6664 of *LNCS*, pages 103–117. Springer, 2011.

[8] T. Bures, P. Hnetynka, and F. Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *Proc. of SERA'06*, pages 40–48. IEEE Computer Society, 2006.

[9] C. Chapman, W. Emmerich, F. Galán Márquez, S. Clayman, and A. Galis. Software Architecture Definition for On-demand Cloud Provisioning. *Cluster Computing*, 15(2):79–100, 2012.

[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C.L. Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.

[11] F. Durán and G. Salaün. Robust Reconfiguration of Cloud Applications. In *Proc. of CBSE'14*, pages 179–184. ACM, 2014.

[12] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL Model Checker. In *Proc. of WRLA'02*, volume 71 of *ENTCS*, pages 115–142. Elsevier, 2002.

[13] X. Etchevers, T. Coupaye, F. Boyer, and N. de Palma. Self-Configuration of Distributed Applications in the Cloud. In *Proc. of CLOUD'11*, pages 668–675. IEEE Computer Society, 2011.

[14] X. Etchevers, G. Salaün, F. Boyer, T. Coupaye, and N. De Palma. Reliable Self-Deployment of Cloud Applications. In *Proc. of SAC'14*, pages 1331–1338. ACM Press, 2014.

[15] J. Fischer, R. Majumdar, and S. Esmaeilsabzali. Engage: A Deployment Management System. In *Proc. of PLDI'12*, pages 263–274. ACM, 2012.

[16] A. Ketfi and N. Belkhatir. A Metamodel-Based Approach for the Dynamic Reconfiguration of Component-Based Software. In *Proc. of ICSR'04*, volume 3107 of *LNCS*, pages 264–273. Springer, 2004.

[17] J. Kramer and J. Magee. Analysing Dynamic Change in Distributed Software Architectures. *IEE Proceedings - Software*, 145(5):146–154, 1998.

[18] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proc. of SIGSOFT FSE'96*, pages 3–14, 1996.

[19] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour Analysis of Software Architectures. In *Proc. of WICSA'99*, volume 140 of *IFIP Conference Proceedings*, pages 35–50. Kluwer, 1999.

[20] J. Matevska-Meyer, W. Hasselbring, and R. Reussner. Software Architecture Description Supporting Component Deployment and System Runtime Reconfiguration. In *Proc. of WCOP'04*, 2004.

[21] N. Medvidovic. ADLs and Dynamic Architecture Changes. In *Proc. of SIGSOFT'96 Workshop*, pages 24–27. ACM, 1996.

[22] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational Abstractions. In *Proc. of CADE'03*, volume 2741 of *LNCS*, pages 2–16. Springer, 2003.

[23] G. Pierre and C. Stratan. ConPaaS: A Platform for Hosting Elastic Cloud Applications. *IEEE Internet Computing*, 16(5):88–92, 2012.

[24] P. Poizat and G. Salaün. Adaptation of Open Component-Based Systems. In *Proc. of FMOODS'07*, volume 4468, pages 141–156. Springer, 2007.

[25] P. Poizat, G. Salaün, and M. Tivoli. On Dynamic Reconfiguration of Behavioural Adaptation. In *Proc. of WCAT'06*, pages 61–69, 2006.

[26] C. M. Poskitt and D. Plump. Hoare-Style Verification of Graph Programs. *Fundam. Inform.*, 118(1-2):135–175, 2012.

[27] G. Salaün, X. Etchevers, N. De Palma, F. Boyer, and T. Coupaye. Verification of a Self-configuration Protocol for Distributed Applications in the Cloud. In *Proc. of SAC'12*, pages 1278–1283. ACM Press, 2012.

[28] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A Graph Based Architectural (Re)configuration Language. In *Proc. of ESEC / SIGSOFT FSE'01*, pages 21–32. ACM Press, 2001.

[29] M. Wirsing, J. Eckhardt, T. Mühlbauer, and J. Meseguer. Design and Analysis of Cloud-Based Architectures with KLAIM and Maude. In *Proc. of WRLA'12*, volume 7571 of *LNCS*, pages 54–82. Springer, 2012.